Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 609 – 627
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Fast and lock-free concurrent priority queues for multi-thread systems☆

Håkan Sundell*, Philippas Tsigas

Department of Computing Science, Chalmers University of Technology and Göteborg University, 412 96 Göteborg, Sweden

Received 29 July 2003; received in revised form 6 December 2004; accepted 14 December 2004

Abstract

We present an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the overall performance of the system. Non-blocking algorithms avoid blocking, and several implementations have been proposed. Previously known non-blocking algorithms of priority queues did not perform well in practice because of their complexity, and they are often based on non-available atomic synchronization primitives. Our algorithm is based on the randomized sequential list structure called Skiplist, and a real-time extension of our algorithm is also described. In our performance evaluation we compare our algorithm with a well-representable set of earlier known implementations of priority queues. The experimental results clearly show that our lock-free implementation outperforms the other lock-based implementations in practical scenarios for 3 threads and more, both on fully concurrent as well as on pre-emptive systems.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Priority queue; Skip list; Non-blocking; Lock-free; Shared data structure; Real-time; Multi-thread; Concurrent

1. Introduction

Priority queues are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components. For example, the ready-queue that is used in the scheduling of tasks in many real-time systems can usually be implemented using a concurrent priority queue. Consequently, the design of efficient implementations of priority queues is an area that has been extensively researched. A priority queue supports two operations, the *Insert* and the *DeleteMin* operation. The abstract definition of a priority queue is a set of key-value pairs, where the key represents a priority. The *Insert* operation inserts a new key-value pair into the set, and the *DeleteMin* operation removes and returns the value of the key-value pair with the lowest key (i.e. highest priority) that was in the set.

To ensure consistency of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [22] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors [23] with the cost of more difficult analysis, although not as efficient on multiprocessor systems [21]) and even starvation.

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause. Lock-free implementations are non-blocking and guarantee that regardless of the contention caused by concurrent operations and the interleaving of their

☆ This is an extended and revised version of the paper with the same title that was presented at IPDPS 2003 and was awarded with the best paper award in the algorithms category.

* Corresponding author.
E-mail addresses: phs@cs.chalmers.se (H. Sundell), tsigas@cs.chalmers.se (P. Tsigas).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.12.005

sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish. This is although different from the type of starvation that could be caused by blocking, where a single operation could block every other operation forever, and cause starvation of the whole system. Wait-free [10] algorithms are lock-free and moreover they avoid starvation as well; in a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Recently, researchers also include obstruction-free [1] implementations to be non-blocking, although this kind of implementation is weaker than lock-free and thus does not guarantee progress of any concurrent operation. Non-blocking algorithms have been shown to be of big practical importance in real applications [30,31], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [26].

There exist several algorithms and implementations of concurrent priority queues. The majority of the algorithms are lock-based, either with a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a small part of the shared data structure. Several different representations of the shared data structure are used, for example: Hunt et al. [14] presents an implementation which is based on heap structures, Grammatikakis et al. [7] compares different structures including cyclic arrays and heaps, and most recently Lotan and Shavit [17] presented an implementation based on the skip list structure [20]. The algorithm by Hunt et al. locks each node separately and uses a technique to scatter the accesses to the heap, thus reducing the contention. Its implementation is publicly available and its performance has been documented on multi-processor systems. Jones [16] also makes use of multiple locks, but implements a fully dynamic tree structure, and tries to only lock the part of the tree necessary at each moment in time. Lotan and Shavit extend the functionality of the concurrent priority queue and assume the availability of a global high-accuracy clock. They apply a lock on each pointer, and as the multi-pointer based skip list structure is used, the number of locks is significantly more than the number of nodes. Its performance has previously only been documented by simulation, with very promising results. The algorithm by Shavit and Zemach [24] is not addressed in this paper, as they implement a bounded [1] priority queue, whereas we address the general priority queues.

Israeli and Rappoport have presented a wait-free algorithm for a concurrent priority queue [15]. This algorithm makes use of strong atomic synchronization primitives [2] that have not been implemented in any currently existing platform. Greenwald has also presented an outline for a lock-free priority queue [8] based on atomic primitives [3] that are not available in modern multiprocessor systems. However, there exists an attempt for a wait-free algorithm by Barnes [2] that uses existing atomic primitives, though this algorithm does not comply with the generally accepted definition of the wait-free property. The algorithm is not yet implemented and the theoretical analysis predicts worse behavior than the corresponding sequential algorithm, which makes it not of practical interest.

One common problem with many algorithms for concurrent priority queues is the lack of precise defined semantics of the operations. It is also seldom that the correctness with respect to concurrency is proved, using a strong property like linearizability [13].

In this paper, we present a lock-free algorithm of a concurrent priority queue that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. Inspired by Lotan and Shavit [17], the algorithm is based on the randomized skip list [20] data structure, but in contrast to [17] it is lock-free. It is also implemented using common synchronization primitives that are available in modern systems. The algorithm is described in detail later in this paper, and the aspects concerning the underlying lock-free memory management are also presented. The precise semantics of the operations are defined and a proof is given that our implementation is lock-free and linearizable. We have performed experiments that compare the performance of our algorithm with a well representative set of earlier implementations of concurrent priority queues known, i.e. the implementation by Lotan and Shavit [17], Hunt et al. [14], and Jones [16]. Experiments were performed on three different platforms, consisting of a multiprocessor system using different operating systems and equipped with 2, 6 or 29 processors. Our results show that our algorithm outperforms the other lock-based implementations in practical scenarios for 3 threads and more, in both highly pre-emptive as well as in fully concurrent environments. We also present an extended version of our algorithm that also addresses certain real-time aspects of the priority queue as introduced by Lotan and Shavit [17].

The rest of the paper is organized as follows. In Section 2 we define the properties of the systems that our implementation is aimed for. The actual algorithm is described in Section 3. In Section 4 we define the precise semantics for the operations on our implementations, as well showing correctness by proving the lock-free and linearizability property. The experimental evaluation that shows the performance of our implementation is presented in Section 5. In Section 6 we extend our algorithm with functionality that can be needed for specific real-time applications. We conclude the paper in Section 7.

[1] The set of possible priorities is restricted.

[2] The algorithm requires ideal implementations of the LL/VL/SC/SC&V/SC2 atomic primitives which only exist as non-efficient software implementations.

[3] The double-word compare-and-swap atomic primitive (CAS2) that can atomically update two arbitrary memory words, has only been implemented in hardware on the 680×0 architectures.

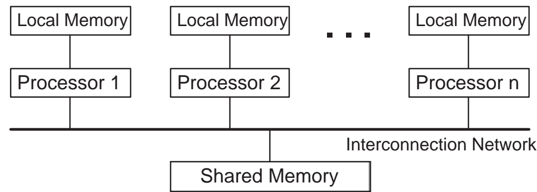

Fig. 1. Shared memory multiprocessor system structure.

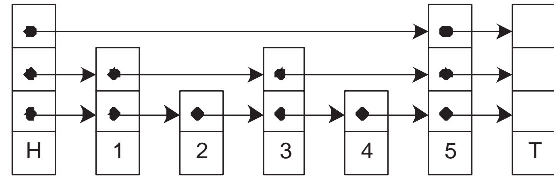

Fig. 2. The skip list data structure with 5 nodes inserted.

2. System description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Fig. 1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to coordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory might not be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

3. Algorithm

The algorithm is based on the sequential skip list data structure invented by Pugh [20]. This structure uses randomization and has a probabilistic time complexity of $O(\log N)$ where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Fig. 2. The maximum height (i.e. the maximum number of next pointers) of the data structure is $\log N$. The height of each inserted node is randomized geometrically in the way that 50% of the nodes should have height 1, 25% of the nodes should have height 2 and so on. To use the data structure as a priority queue, the nodes are ordered in respect of priority (which has to be unique for each node [4]), the nodes with highest priority are located first in the list. The fields of each node item are described in Fig. 3 as it is used in this implementation. For all code examples in this paper, code that is inside of small framed boxes are only used for the special real-time [5] version of our implementation that involves timestamps (see Section 6), and are thus not included in the standard version of the implementation.

```
union Link
        : word
        ⟨p,d⟩: ⟨pointer to Node,boolean⟩

union VLink
        : word
        ⟨p,d⟩: ⟨pointer to Value,boolean⟩

structure Node
        key,level,validLevel [,timeInsert] : integer
        value : union VLink
        next[level]: union Link
        prev : pointer to Node

// Global variables
head, tail : pointer to Node
// Local variables (for all functions/procedures)
newNode, savedNodes[maxlevel] : pointer to Node
node1, node2, prev, last : pointer to Node
i, level : integer

function CreateNode(level:integer, key:integer,
 value:pointer to Value):pointer to Node
C1     node:=MALLOC_NODE();
C2     node.prev:=NULL;
C3     node.validLevel:=0;
C4     node.level:=level;
C5     node.key:=key;
C6     node.value:=⟨value,false⟩;
C7     return node;
```

Fig. 3. The basic algorithm details.

In order to make the skip list construction concurrent and non-blocking, we are using three of the standard atomic synchronization primitives, Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Fig. 4 describes the specification of these primitives which are available in most modern platforms.

To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. Our solution is to have additional information in each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partially deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted. However, the node is interpreted to be inserted in the abstract sense already when inserted at the lowest level. Thus, insert and delete operations always start with updating

[4] In order to assign several objects the same priority, this limitation can be overcome by building the priority (key) so that only some bits represent the real priority and remaining bits are chosen in order to achieve uniqueness.

[5] In the sense that DeleteMin operations can only return items that were fully inserted before the start of the DeleteMin operation.

```
function TAS(value:pointer to word):boolean
    atomic do
        if *value=0 then
            *value:=1;
            return true;
        else return false;

procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word,
 newvalue:word):boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

Fig. 4. Definitions of the Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

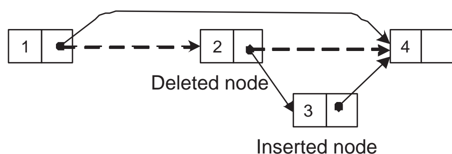

Fig. 5. Concurrent inserting and deleting operations can accidentally delete both nodes.

the lowest level first, and then continue with the remaining consecutive levels. In order to improve the performance of concurrent operations, we have chosen for the insert operation to update the remaining levels starting from the second lowest level going upwards, and the opposite direction was chosen for the delete operation.

One problem, that is general for non-blocking implementations that are based on the linked-list structure, arises when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted—then the new node will be deleted as well, as illustrated in Fig. 5. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any modern multiprocessor system. A second solution is to insert auxiliary nodes [32] between each two normal nodes, and the latest method introduced by Harris [9] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, therefore bits 0 and 1 of the address are always set to zero. The method is then to use the previously unused bit 0 of the next pointer to mark that this node is about to be deleted, using CAS. Any concurrent *Insert* operation will then be notified about the deletion, when its CAS operation will fail.

3.1. The basic steps of the algorithm

The main algorithm steps, for inserting a new node at an arbitrary position in our skip list will thus be like follows: (I) find the appropriate position (i.e. previous nodes) for inserting the new node, (II) atomically update the next pointers of the to-be-previous nodes starting with the lowest level and continuing with the remaining levels in consecutive order.

The main steps of the algorithm for deleting a node at an arbitrary position are the following: (I) find the appropriate position (i.e. the node and previous nodes) for deleting, (II) set the main deletion indication on the to-be-deleted node, (III) set the deletion marks on the next pointers of the to-be-deleted node starting with the lowest level and continuing with the remaining levels in consecutive order, (IV) atomically update the next pointers of the previous nodes of the to-be-deleted node starting with the lowest level and continuing with the remaining levels in consecutive order from the topmost level. As will be shown later in the detailed description of the algorithm, helping techniques need to be applied in order to achieve the lock-free property, following the same steps as the main algorithm for inserting and deleting.

3.2. Memory management

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is (or will be) traversing that node. For efficiency reasons we also need to be able to trust the prev pointers of deleted nodes, as we would otherwise be forced to re-start the traversing from the head node whenever reaching a deleted node while traversing and possibly incur severe performance penalties. This need is especially important for operations that try to help other delete operations in progress. Our demands on the memory management therefore rules out the SMR or ROP methods by Michael [18] and Herlihy et al. [12], respectively, as they can only guarantee a limited number of nodes to be safe, and these guarantees are also related to individual threads and never to an individual node structure. However, stronger memory management schemes as for example reference counting would be sufficient for our needs. There exists a general lock-free reference counting scheme by Detlefs et al. [3], though based on the non-available CAS2 atomic primitive.

For our implementation, we selected the lock-free memory management scheme invented by Valois [32] and corrected by Michael and Scott [19], which makes use of the FAA and CAS atomic synchronization primitives. Using this scheme we can assure that a node can only be reclaimed when there is no prev pointer in the list or a local pointer variable that points to it. A general problem with reference counting, is that it can not handle cyclic garbage (i.e. 2 or

more nodes that should be recycled but reference each other, and therefore each node keeps a positive reference count, although they are not referenced by the main structure). However, this is not a problem with our solution, as we only reference count prev pointers, and not the next pointers as we never need to traverse the next pointers of a deleted node.

The memory management scheme should also support means to de-reference pointers safely. If we simply de-reference a next pointer using the means of the programming language, it might be that the corresponding node has been reclaimed before we could access it. It can also be that the deletion mark that is connected to the next pointer was set, thus marking that the node is deleted. The scheme by Valois et al. supports lock-free pointer de-referencing and can easily be adopted to handle deletion marks.

The following functions are defined for safe handling of the memory management:

```
function MALLOC_NODE() :pointer to Node
function READ_NODE(address:pointer to Link) :pointer to Node
function COPY_NODE(node:pointer to Node) :pointer to Node
procedure RELEASE_NODE(node:pointer to Node)
```

The functions *READ_NODE* atomically de-references the given link and increases the reference counter for the corresponding node. In case the deletion mark of the link is set, the *READ_NODE* function then returns NULL. The function *MALLOC_NODE* allocates a new node from the memory pool and returns a corresponding pointer with a reference count of one. The function *RELEASE_NODE* decrements the reference counter on the corresponding given node. If the reference counter reaches zero, the function will recursively call *RELEASE_NODE* on the nodes that this node has owned pointers to (i.e. the prev pointer), and then it reclaims the node. The *COPY_NODE* function increases the reference counter for the corresponding given node.

As the details of how to efficiently apply the memory management scheme to our basic algorithm are not always trivial, we will provide a detailed description of them together with the detailed algorithm description in this section.

3.3. Traversing

While traversing the nodes, processes will eventually reach nodes that are marked to be deleted. As the process that invoked the corresponding delete (e.g. *DeleteMin*) operation might be pre-empted, that delete operation has to be helped to finish before the traversing process can continue. However, it is only necessary to help the part of the delete operation on the current level in order to be able to traverse to the next node. The function *ReadNext*, see Fig. 6, traverses to the next node of *node1* on the given level while helping (and then sets *node1* to the previous node of the helped one) any marked nodes in between to finish the deletion. The function *ScanKey*, see Fig. 6, traverses in several steps through the next pointers (starting from *node1*)

```
function ReadNext(node1:pointer to pointer to Node,
 level:integer):pointer to Node
R1    if (*node1).value.d=true then
R2        *node1:=HelpDelete(*node1,level);
R3    node2:=READ_NODE((*node1).next[level]);
R4    while node2=NULL do
R5        *node1:=HelpDelete(*node1,level);
R6        node2:=READ_NODE((*node1).next[level]);
R7    return node2;

function ScanKey(node1:pointer to pointer to Node,
 level:integer, key:integer):pointer to Node
S1    node2:=ReadNext(node1,level);
S2    while node2.key < key do
S3        RELEASE_NODE(*node1);
S4        *node1:=node2;
S5        node2:=ReadNext(node1,level);
S6    return node2;
```

Fig. 6. Functions for traversing and searching for nodes in the skip list data structure.

at the current level until it finds a node that has the same or higher key (priority) value than the given key. It also sets *node1* to be the previous node of the returned node.

3.4. Inserting and deleting nodes

The implementation of the *Insert* operation, see Fig. 7, starts in lines I2–I4 with creating the new node (*newNode*) and choosing its height (*level*) by calling the *randomLevel* function. This function roughly simulates a repeated coin tossing, counting the number of times (up to the maximum level) the upper (or lower if that was chosen) side of the coin turns up from the start of the function, thus giving the distribution associated with the skip list [20]. In lines I5–I11 the implementation continues with a search phase to find the node after which *newNode* should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found (*node1*). When going down one level, the last node traversed on that level is remembered (*savedNodes*) for later use (this is where we should insert the new node at that level). Now it is possible that there already exists a node with the same priority as of the new node, this is checked in lines I12–I24, the value of the old node (*node2*) is changed atomically with a CAS. Otherwise, in lines I25–I42 it starts trying to insert the new node starting with the lowest level and increasing up to the level of the new node. The next pointers of the nodes (to become previous) are changed atomically with a CAS. After the new node has been inserted at the lowest level, it is possible that it is deleted by a concurrent delete (e.g. *DeleteMin*) operation before it has been inserted at all levels, and this is checked in lines I38 and I44.

The *RemoveNode* procedure, see Fig. 8, removes the given *node* from the linked list structure at the given *level*, using a given hint *prev* for the previous node. It first searches for the right position of the previous node according to the

```
function Insert(key:integer, value:pointer to Value):
  boolean
I1       [TraverseTimeStamps();]
I2      Choose level randomly according to the skip list
          distribution
I3      newNode:=CreateNode(level,key,value);
I4      COPY_NODE(newNode);
I5      node1:=COPY_ NODE(head);
I6      for i:=maxLevel-1 to 1 step -1 do
I7          node2:=ScanKey(&node1,i,key);
I8          RELEASE_NODE(node2);
I9          if i< level then savedNodes[i]:=
              COPY_ NODE(node1);
I10     while true do
I11         node2:=ScanKey(&node1,0,key);
I12         ⟨value2,d⟩:=node2.value;
I13         if d=false and node2.key=key then
I14             if CAS(&node2.value,⟨value2,false⟩,
                  ⟨value,false⟩) then
I15                 RELEASE_NODE(node1);
I16                 RELEASE_NODE(node2);
I17                 for i:=1 to level-1 do
I18                     RELEASE_NODE(savedNodes[i]);
I19                 RELEASE_NODE(newNode);
I20                 RELEASE_NODE(newNode);
I21                 return true2;
I22             else
I23                 RELEASE_NODE(node2);
I24                 continue;
I25         newNode.next[0]:= ⟨node2,false⟩,
I26         RELEASE_NODE(node2);
I27         if CAS(&node1.next[0],⟨node2,false⟩,
              ⟨newNode,false⟩) then
I28             RELEASE_NODE(node1);
I29             break;
I30         Back-Off
I31     for i:=1 to level-1 do
I32         newNode.validLevel:=i;
I33         node1:=savedNodes[i];
I34         while true do
I35             node2:=ScanKey(&node1,i,key);
I36             newNode.next[i]:=⟨node2,false⟩;
I37             RELEASE_NODE(node2);
I38             if newNode.value.d=true or
                  CAS(&node1.next[i],node2,newNode) then
I39                 RELEASE_NODE(node1);
I40                 break;
I41             Back-Off
I42     newNode.validLevel:=level;
I43         [newNode.timeInsert:=getNextTimeStamp();]
I44     if newNode.value.d=true then newNode:=
          HelpDelete(newNode,0);
I45     RELEASE_ NODE(newNode);
I46     return true;
```

Fig. 7. The algorithm for the *Insert* operation that inserts a new node into the skip list data structure.

```
procedure RemoveNode(node: pointer to Node,
  prev: pointer to pointer to Node, level:integer)
RN1   while true do
RN2       if node.next[level]=⟨NULL,true⟩ then break;
RN3       last:=ScanKey(prev,level,node.key);
RN4       RELEASE_NODE(last);
RN5       if last≠node or node.next[level]=⟨NULL,true⟩
            then break;
RN6       if CAS(&(*prev).next[level],⟨node,false⟩,
            ⟨node.next[level].p,false⟩) then
RN7           node.next[level]:=⟨NULL,true⟩;
RN8           break;
RN9       if node.next[level]=⟨NULL,true⟩ then break;
RN10      Back-Off
```

Fig. 8. The algorithm for the *RemoveNode* procedure that removes an arbitrary node in the skip list data structure at a given level.

```
function DeleteMin():pointer to Node
D1     [TraverseTimeStamps();]
D2     [time:=getNextTimeStamp();]
D3    prev:=COPY_NODE(head);
D4    while true do
D5        node1:=ReadNext(&prev,0);
D6        if node1=tail then
D7            RELEASE_ NODE(prev);
D8            RELEASE_ NODE(node1);
D9            return NULL;
      retry:
D10       ⟨value,d⟩:=node1.value;
D11       if node1 ≠ prev.next[0].p then
D12           RELEASE_ NODE(node1);
D13           continue;
D14       if d=false [and compareTimeStamp(time,
            node1.timeInsert)> 0] then
D15           if CAS(&node1.value,⟨value,false⟩,⟨value,
                true⟩) then
D16               node1.prev:=prev;
D17               break;
D18           else goto retry;
D19       else d=true then
D20           node1:=HelpDelete(node1,0);
D21       RELEASE_NODE(prev);
D22       prev:=node1;
D23   for i:=0 to node1.level-1 do
D24       repeat
D25           ⟨node2,d⟩:=node1.next[i];
D26       until d= true or CAS(&node1.next[i],⟨node2,false⟩,
            ⟨node2,true⟩);
D27   prev:=COPY_NODE(head);
D28   for i:=node1.level-1 to 0 step -1 do
D29       RemoveNode(node1,&prev,i);
D30   RELEASE_NODE(prev);
D31   RELEASE_NODE(node1);
D32   RELEASE_NODE(node1); /* Delete the node */
D33   return value;
```

Fig. 9. The algorithm for the *DeleteMin* operation that removes the first node in the skip list data structure.

key of *node* in line RN3. It verifies in line RN5 that *node* is still part of the linked list structure at the present level. In line RN6 it tries to remove *node* by changing the next pointer of the previous node using the CAS sub-operation. If the CAS failed, possibly because of concurrent changes to the *prev* node, the whole procedure retries. As this procedure can be invoked concurrently on the same node argument, it synchronizes with the possibly other invocations in lines RN2, RN5, RN7 and RN9 in order to avoid executing sub-operations that have already been performed.

The *DeleteMin* operation, see Fig. 9, starts from the head node and finds the first node (*node*1) in the list that

```
function HelpDelete(node:pointer to Node,
 level:integer):pointer to Node
H1   for i:=level to node.level-1 do
H2     repeat
H3       ⟨node2,d⟩:=node.next[i];
H4     until d= true or CAS(&node.next[i],⟨node2,false⟩,
          ⟨node2,true⟩);
H5   prev:=node.prev;
H6   if not prev or level ≥prev.validLevel then
H7     prev:=COPY_NODE(head);
H8     for i:=maxLevel-1 to level step -1 do
H9       node2:=ScanKey(&prev,i,node.key);
H10      RELEASE_NODE(node2);
H11  else COPY_NODE(prev);
H12  RemoveNode(node,&prev,level);
H13  RELEASE_NODE(node);
H14  return prev;
```

Fig. 10. The algorithm for the *HelpDelete* procedure that help concurrent delete operations to finish.

does not have its deletion mark on the value set, see lines D3–D15. It tries to set this deletion mark in line D15 using the CAS primitive, and if it succeeds it also writes a valid pointer to the prev field of the node. This prev field is necessary in order to increase the performance of concurrent *HelpDelete* functions, these operations otherwise would have to search for the previous node in order to complete the deletion. The next step is to mark the deletion bits of the next pointers in the node, starting with the lowest level and going upwards, using the CAS primitive in each step, see lines D23–D26. Afterwards in lines D27–D29 it starts the actual deletion by calling the *RemoveNode* procedure, starting at the highest level and continuing downwards. The reason for doing the deletion in decreasing order of levels is that concurrent search operations also start at the highest level and proceed downwards, in this way the concurrent search operations will sooner avoid traversing this node.

3.5. Helping

The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock-free property (that at least one thread is doing progress) on pre-emptive systems, whenever a search operation finds a node that is about to be deleted, it calls the *HelpDelete* function and then proceeds searching from the previous node of the deleted. The *HelpDelete* function, see Fig. 10, tries to fulfill the deletion on the current level and returns a reference to the previous node when the deletion is completed. It starts in lines H1–H4 with setting the deletion mark on all next pointers in case they have not been set. In lines H5–H6 it checks if the node given in the prev field is valid for deletion on the current level, otherwise it searches for the correct previous node (*prev*) in lines H7–H10. The actual deletion of this node on the current level takes place in line H12 by calling the *RemoveNode* procedure.

In fully concurrent systems though, the helping strategy can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed attempts to help concurrent *DeleteMin* operations that hinders the progress of the current operation, puts the operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is proportional to the number of threads, and for each consecutive entering of back-off mode during one operation invocation, the duration is increased exponentially.

3.6. Use with other memory management schemes

As described earlier we have chosen the lock-free reference counting method by Valois [32] and Michael and Scott [19], because this was the only practical and lock-free garbage collection scheme that could enable the algorithm to avoid restarts from the head node when traversing through the skip list. This feature is very important for scalability with respect to increasing concurrency, as the number of concurrent deletions of nodes in the skip list will then also increase.

However, if we can sacrifice this feature of our algorithm that allows better scalability, and instead always restart from the head node when reaching a node marked as deleted while traversing, other memory management schemes could be applied. As we then no longer would need to be able to guarantee safety of the prev pointer of a deleted node, the SMR [18] or ROP [12] schemes could be applied. The hazard pointer style of garbage collection can guarantee that all local pointers to nodes that each process holds will not be reclaimed for reuse until the corresponding hazard pointer is cleared.

The needed main modification of our algorithm would simply be to remove line D16, so that the prev pointer is never set. All local variables that point to nodes, including the savedNodes array, need to be associated with corresponding hazard pointers. Thus, the interface to the new memory management is done as before through the *MALLOC_NODE*, *READ_NODE*, *COPY_NODE* and *RELEASE_NODE* functions. The *READ_NODE* function should de-reference a pointer and set the corresponding hazard pointer, unless the pointer was marked when the function instead returns NULL. The *COPY_NODE* function could be implemented by copying the corresponding hazard pointer to another one, though most of the time it can be optimized to be obsolete. The *RELEASE_NODE* function is simply implemented by clearing the corresponding hazard pointer, and subsequent calls can mostly be optimized to be obsolete. Care must be taken for functions that returns safe pointers, such as *HelpDelete*, so that the calling functions know which particular hazard pointer that is associated with the corresponding pointer.

3.7. Related work with skip lists

This paper describes the first[6] lock-free algorithm of a skip list data structure. Very similar constructions have appeared in the literature afterwards, by Fraser [6], Fomitchev [4] and Fomitchev and Ruppert [5]. As both Fraser's and Fomitchev's constructions appeared quite some time later in the literature than ours, it was not possible to compare them in our original publications. However, we have recently studied the other's approaches and have found some significant differences, although the main ideas are essentially the same. The differences are mainly related to performance issues:

- Compared to Fraser's approach, our skip list construction does not suffer from possible restarts of the full search phase from the head level when reaching a deleted node, as our nodes also contains a backlink pointer that is set at the time of deletion. This enables us to step one step backwards when reaching a deleted node, and to directly remove the deleted node. Both Fraser's and our construction uses arrays for remembering positions, though Fraser unnecessarily remembers also the successor on each level which could incur performance penalties through the garbage collector used.
- Compared to Fomitchev's and Fomitchev and Ruppert's approach, their construction does not use an array for remembering positions, which forces their construction to perform two full search phases when inserting or deleting nodes. In addition to have backlink pointers in order to be able to step back when reaching a deleted node, their construction also uses an extra pointer mark that is set (using an extra and expensive CAS operation) on the predecessor node in order to earlier notify concurrent operations of the helping duty. In our construction we only have one backlink pointer for all levels of a node, because of a performance trade-off between the usefulness for helping operations and the cost that keeping extra pointers could incur for the garbage collection.

4. Correctness

In this section we present the proof of our algorithm. We first prove that our algorithm is linearizable [13] and then that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 1. We denote with L_t the abstract internal state of a priority queue at the time t. L_t is viewed as a set of pairs $\langle p, v\rangle$ consisting of a unique priority p and a corresponding value v. The operations that can be performed on the priority queue are Insert (I) and DeleteMin (DM). The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. The return value of $true_2$ is returned by an Insert operation that has succeeded to update an existing node, the return value of *true* is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:

$$\langle p_1, _\rangle \notin L_{t_1} : I_1(\langle p_1, v_1\rangle) = true,$$
$$L_{t_2} = L_{t_1} \cup \{\langle p_1, v_1\rangle\}, \quad (1)$$

$$\langle p_1, v_{1_1}\rangle \in L_{t_1} : I_1(\langle p_1, v_{1_2}\rangle) = true_2,$$
$$L_{t_2} = L_{t_1} \setminus \{\langle p_1, v_{1_1}\rangle\} \cup \{\langle p_1, v_{1_2}\rangle\}, \quad (2)$$

$$\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\}$$
$$: DM_1() = \langle p_1, v_1\rangle,\ L_{t_2} = L_{t_1} \setminus \{\langle p_1, v_1\rangle\}, \quad (3)$$

$$L_{t_1} = \emptyset : DM_1() = \bot. \quad (4)$$

Definition 2. In a global time model each concurrent operation Op "occupies" a time interval $[b_{Op}, f_{Op}]$ on the linear time axis ($b_{Op} < f_{Op}$). The precedence relation (denoted by '$\rightarrow$') is a relation that relates operations of a possible execution, $Op_1 \rightarrow Op_2$ means that Op_1 ends before Op_2 starts. The precedence relation is a strict partial order. Operations incomparable under $\rightarrow$ are called *overlapping*. The overlapping relation is denoted by $\|$ and is commutative, i.e. $Op_1 \| Op_2$ and $Op_2 \| Op_1$. The precedence relation is extended to relate sub-operations of operations. Consequently, if $Op_1 \rightarrow Op_2$, then for any sub-operations op_1 and op_2 of Op_1 and Op_2, respectively, it holds that $op_1 \rightarrow op_2$. We also define the direct precedence relation $\rightarrow_d$, such that if $Op_1 \rightarrow_d Op_2$, then $Op_1 \rightarrow Op_2$ and moreover there exists no operation Op_3 such that $Op_1 \rightarrow Op_3 \rightarrow Op_2$.

Definition 3. In order for an implementation of a shared concurrent data object to be linearizable [13], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.

[6] Our results were submitted for reviewing in October 2002 and published as a technical report [27] in January 2003. It was officially published in April 2003 [28], receiving a best paper award, and an extended version was also published in March 2004 [29]. Very similar constructions have appeared in the literature afterwards, by Fraser in February 2004 [6], Fomitchev in November 2003 [4] and Fomitchev and Ruppert in July 2004 [5].

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 4. The pair $\langle p, v\rangle$ is present ($\langle p, v\rangle \in L$) in the abstract internal state L of our implementation, when there is a next pointer from a present node on the lowest level of the skip list that points to a node that contains the pair $\langle p, v\rangle$, and this node is not marked as deleted with the mark on the value.

Lemma 1. *The definition of the abstract internal state for our implementation is consistent with all concurrent operations examining the state of the priority queue.*

Proof. As the next and value pointers are changed using the CAS operation, we are sure that all threads see the same state of the skip list, and therefore all changes of the abstract internal state will appear to be atomic. □

Definition 5. The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations after the decision point, the operation will have the same result. We define the verification point of an operation to be the atomic statement where a sub-state of the priority queue is read, and this sub-state is verified to have certain properties before the passing of the decision point. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.

We will now use these points in order to show the existence and location in execution history of a point where the concurrent operation can be viewed as it occurred atomically, i.e. the *linearizability point*.

Lemma 2. *An* Insert *operation which succeeds* ($I(\langle p, v\rangle) = true$), *takes effect atomically at one statement.*

Proof. The decision point for an Insert operation which succeeds ($I(\langle p, v\rangle) = true$), is when the CAS sub-operation in line I27 (see Fig. 7) succeeds, all following CAS sub-operations will eventually succeed, and the Insert operation will finally return *true*. The state of the list (L_{t_1}) directly before the passing of the decision point must have been $\langle p, _\rangle \notin L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v\rangle \in L_{t_2}$. Consequently, the linearizability point will be the CAS sub-operation in line I27, as this statement will then match the semantics in Eq. (1). □

Lemma 3. *An* Insert *operation which updates* ($I(\langle p, v\rangle) = true_2$), *takes effect atomically at one statement.*

Proof. The decision point for an Insert operation which updates ($I(\langle p, v\rangle) = true_2$), is when the CAS will succeed in line I14. The state of the list (L_{t_1}) directly before passing the decision point must have been $\langle p, _\rangle \in L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v\rangle \in L_{t_2}$. Consequently, the linearizability point will be the CAS sub-operation in line I14, as this statement will then match the semantics in Eq. 2. □

Lemma 4. *A* DeleteMin *operations which fails* ($DM() = \bot$), *takes effect atomically at one statement.*

Proof. The decision point for an DeleteMin operations which fails ($DM() = \bot$), is when the hidden read sub-operation of the *ReadNext* sub-operation in line D5 successfully reads the next pointer on lowest level that equals the tail node. The state of the list (L_t) directly before the passing of the decision point must have been $L_t = \emptyset$. Consequently, the linearizability point will be the hidden read sub-operation of the next pointer in line D5, as this statement will then match the semantics in Eq. (4). □

Lemma 5. *A* DeleteMin *operation which succeeds* ($DM() = \langle p_1, v_1\rangle$ *where* $\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\}$), *takes effect atomically at one statement.*

Proof. The decision point for an DeleteMin operation which succeeds is when the CAS sub-operation in line D15 (see Fig. 9) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle p_1, v_1\rangle \in L_{t_3}$, otherwise the CAS would have failed. The state of the list directly after passing the CAS sub-opertion in line D15 (i.e. the state-change point) will be $\langle p, _\rangle \notin L_{t_4}$. The state of the list at the time of the read sub-operation of the next pointer in D11 (i.e. the verification point) must have been $\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\}$. Unfortunately this does not completely match the semantic definition of the operation in Eq. (3).

However, none of the other concurrent operations linearizability points is dependent on the to-be-deleted node's state as marked or not marked during the time interval from the verification point to the state-change point. Clearly, the linearizability points of Lemma 2 is independent during this time interval, as the to-be-deleted node must be different from the corresponding $\langle p, v\rangle$ term, as Lemma 2 views the to-be-deleted node as present during the time interval. The linearizability point of Lemma 3 is independent during the time interval, as the to-be-deleted node must be different from the corresponding $\langle p, v\rangle$ term, otherwise the CAS sub-operation in line D15 of this operation would have failed.

The linearizability point of Lemma 4 is independent, as that linearizability point depends on the head node's next pointer pointing to the tail node or not. Finally, the linearizability point of this lemma is independent, as the to-be-deleted node would be different from the corresponding $\langle p_1, v_1\rangle$ term, otherwise the CAS sub-operation in line D15 of this operation invocation would have failed.

Therefore all together, we could safely interpret the to-be-deleted node to be not present already directly after passing the verification point ($\langle p, _\rangle \notin L_{t_2}$). Consequently, the linearizability point will be the read sub-operation of the next pointer in line D11, as this statement will then match the semantics in Eq. (3). □

Definition 6. We define the relation $\Rightarrow$ as the total order and the relation $\Rightarrow_d$ as the direct total order between all operations in the concurrent execution. In the following formulas, $E_1 \implies E_2$ means that if E_1 holds then E_2 holds as well, and $\oplus$ stands for exclusive or (i.e. $a \oplus b$ means $(a \vee b) \wedge \neg(a \wedge b)$):

$$Op_1 \rightarrow_d Op_2, \nexists Op_3.Op_1 \Rightarrow_d Op_3, \nexists Op_4.Op_4 \Rightarrow_d Op_2 \implies Op_1 \Rightarrow_d Op_2 \tag{5}$$

$$Op_1 \parallel Op_2 \implies Op_1 \Rightarrow_d Op_2 \oplus Op_2 \Rightarrow_d Op_1, \tag{6}$$

$$Op_1 \Rightarrow_d Op_2 \implies Op_1 \Rightarrow Op_2, \tag{7}$$

$$Op_1 \Rightarrow Op_2, Op_2 \Rightarrow Op_3 \implies Op_1 \Rightarrow Op_3. \tag{8}$$

Lemma 6. *The operations that are directly totally ordered using formula* 5, *form an equivalent valid sequential execution.*

Proof. If the operations are assigned their direct total order ($Op_1 \Rightarrow_d Op_2$) by formula 5 then also the linearizability points of Op_1 are executed before the respective points of Op_2. In this case the operations semantics behave the same as in the sequential case, and therefore all possible executions will then be equivalent to one of the possible sequential executions. □

Lemma 7. *The operations that are directly totally ordered using formula* 6 *can be ordered unique and consistent, and form an equivalent valid sequential execution.*

Proof. Assume we order the overlapping operations according to their linearizability points. As the state before as well as after the linearizability points is identical to the corresponding state defined in the semantics of the respective sequential operations in Eqs. (1)–(4), we can view the operations as occurring at the linearizability point. As the linearizability points consist of atomic operations and are therefore ordered in time, no linearizability point can occur at the very same time as any other linearizability point, therefore giving a unique and consistent ordering of the overlapping operations. □

Lemma 8. *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof. We now examine the possible execution paths of our implementation. There are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct position etc.), the retry-loops take place when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding *Insert* or *DeleteMin* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 1. *The algorithm implements a lock-free and linearizable priority queue.*

Proof. Following from Lemmas 6 and 7 and using the direct total order we can create an identical (with the same semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 3, the implementation is therefore linearizable. As the semantics of the operations are basically the same as in the skip list [20], we could use the corresponding proof of termination. This together with Lemma 8 and that the state is only changed at one atomic statement (Lemmas 1, 2, 3, 5, 4), gives that our implementation is lock-free. □

5. Experiments

We have performed experiments using 1 up to 28 threads on three different platforms, each with different levels of real concurrency, architecture and operating system. Besides our implementation, we also performed the same experiments with four lock-based implementations. These are; (1) a single-lock protected skip list, (2) the implementation using multiple locks and skip lists by Lotan and Shavit [17],

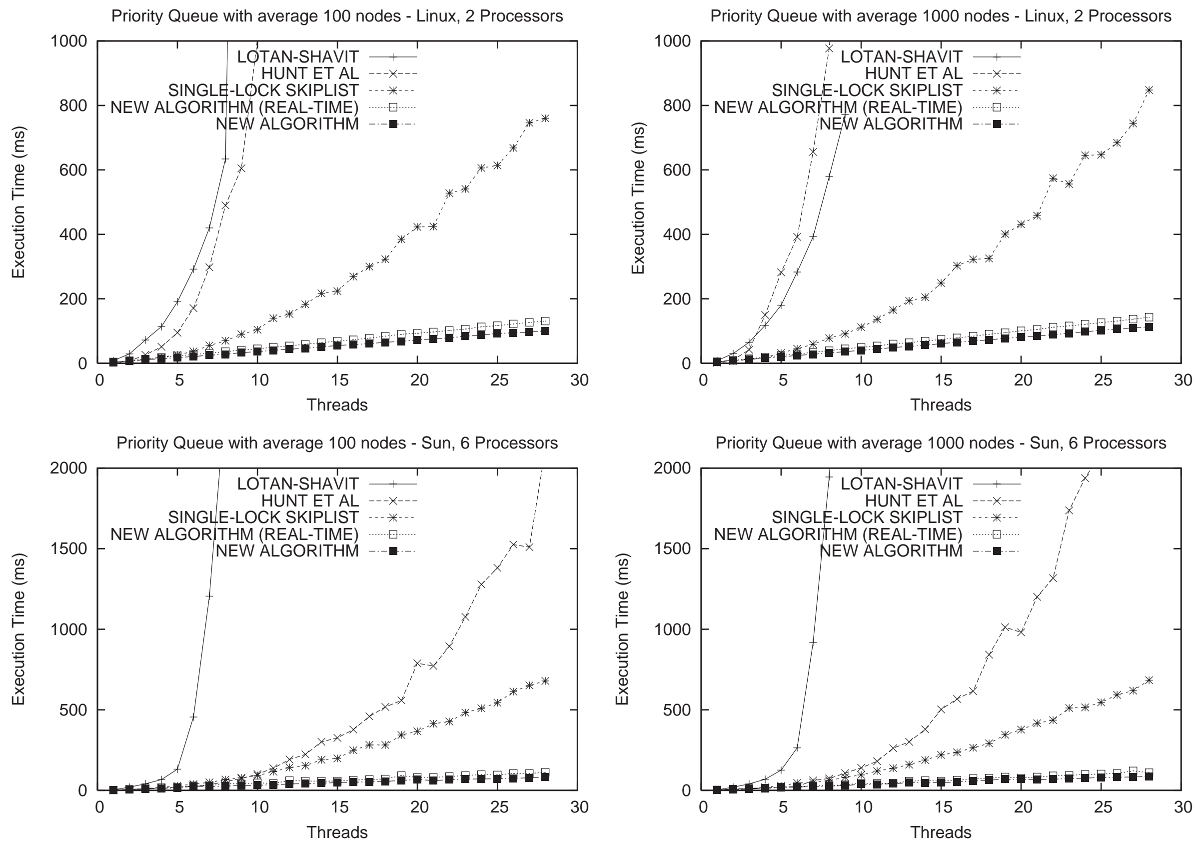

Fig. 11. Experiment with priority queues and low or medium concurrency, with initial 100 or 1000 nodes, using spinlocks for mutual exclusion.

(3) the heap-based implementation using multiple locks by Hunt et al. [14], and (4) the tree-based implementation by Jones [16]. As Lotan and Shavit implements the real-time properties as presented in Section 6 but Hunt et al. does not, we used both the ordinary as well the real-time version of our implementation.

The key values of the inserted nodes are randomly chosen between 0 and $1{,}000{,}000 * n$, where n is the number of threads. Each experiment is repeated 50 times, and an average execution time for each experiment is estimated. Exactly the same sequential operations are performed for all different implementations compared. A clean-cache operation is performed just before each sub-experiment. All implementations are written in C and compiled with the highest optimization level, except from the atomic primitives, which are written in assembler.

5.1. Low or medium concurrency

To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor Pentium II 450 MHz PC running Linux. A set of experiments was also performed on a Sun Ultra 880 with 6 processors running Solaris 9. In our experiments each concurrent thread performs 10,000 sequential operations, randomly chosen with a distribution of 50% *Insert* operations versus 50% *DeleteMin* operations. The dictionaries are initialized with 100 or 1000 nodes before the start of the experiments. The implementation by Jones uses system semaphores for the mutual exclusion. All other lock-based implementations were evaluated using simple spin-locks, [7] and as well using system semaphores. The results from these experiments are shown in Fig. 11 for the spinlock-based implementations and in Fig. 12 for the semaphore-based, both together with the new lock-free implementation. The average execution time is drawn as a function of the number of threads.

5.2. Full concurrency

In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 250 MHz system running Irix 6.5 with 29 processors. With the exception for the implementation by Jones which can only use semaphores, all lock-based implementations were only evaluated using simple spin-locks, as those are always more efficient than

[7] The spin-locks used in our experiments are of the "Test and Test-And-Set" type, which means that they only execute the TAS atomic primitive after having read the lock variable to be zero. In common shared memory architectures as such based on cache coherence protocols, this means that the spinning will normally be done locally on the cache contents.

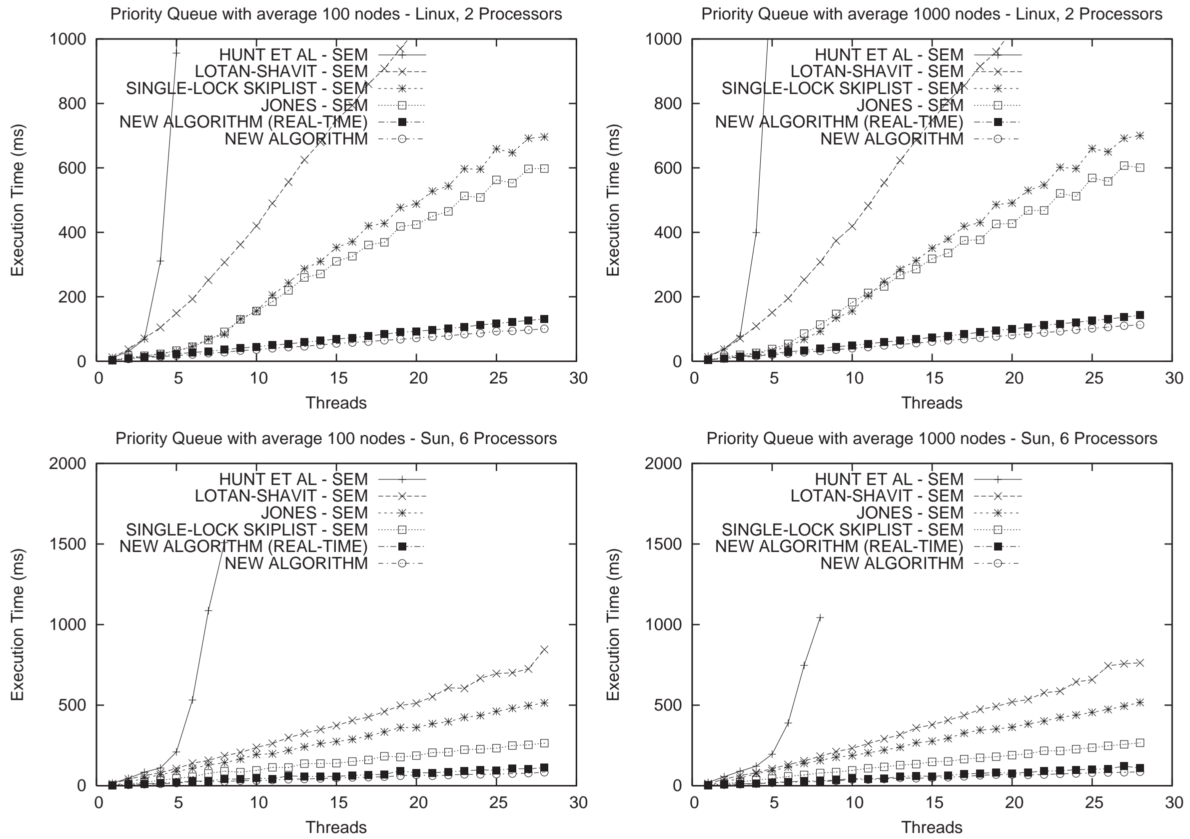

Fig. 12. Experiment with priority queues and low or medium concurrency, with initial 100 or 1000 nodes, using semaphores for mutual exclusion.

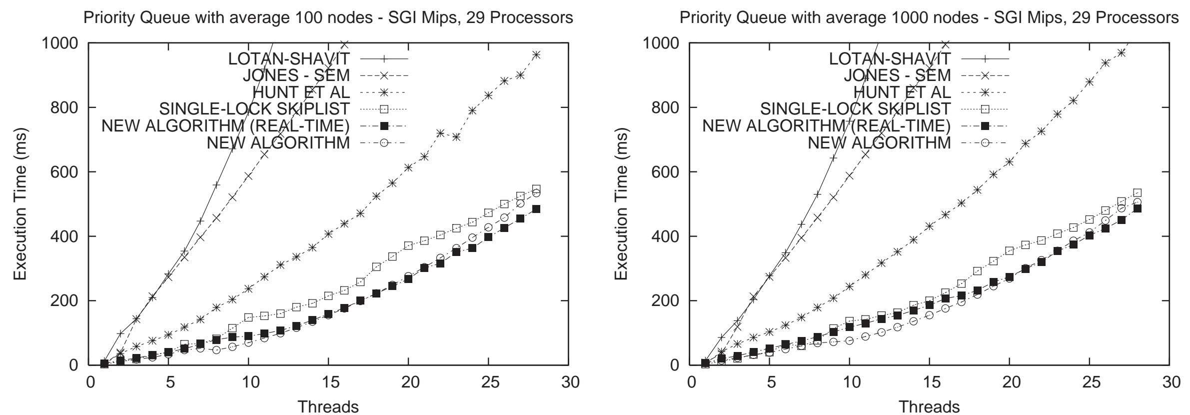

Fig. 13. Experiment with priority queues and full concurrency, running with average 100 or 1000 nodes.

semaphores on fully concurrent systems. In the first experiments each concurrent thread performs 10,000 sequential operations, randomly chosen with a distribution of 50% *Insert* operations versus 50% *DeleteMin* operations, operating on dictionaries that are initialized with 100 or 1000 nodes. The results from these experiments are shown in Fig. 13. The average execution time is drawn as a function of the number of threads.

For the two implementations of our algorithm we also ran experiments, where the dictionaries were initialized with 100, 200, 500, 1000, 2000, 5000 or 10,000 nodes. The results from these experiments are shown in Fig. 14. The average execution time is drawn as a function of the number of threads.

In order to examine the scalability with respect to size in the non contended or low concurrency situations for all included implementations, we ran experiments with 1 or 2 threads, where the dictionaries were initialized with 100, 200, 500, 1000, 2000, 5000 or 10,000 nodes. The results from these experiments are shown in Fig. 15. The average

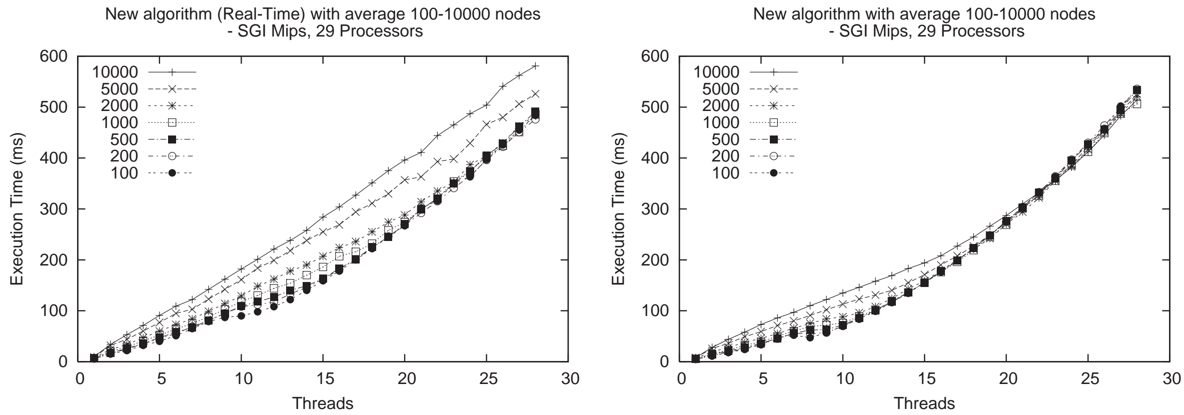

Fig. 14. Experiment with priority queues and full concurrency, running with average 100–10,000 nodes.

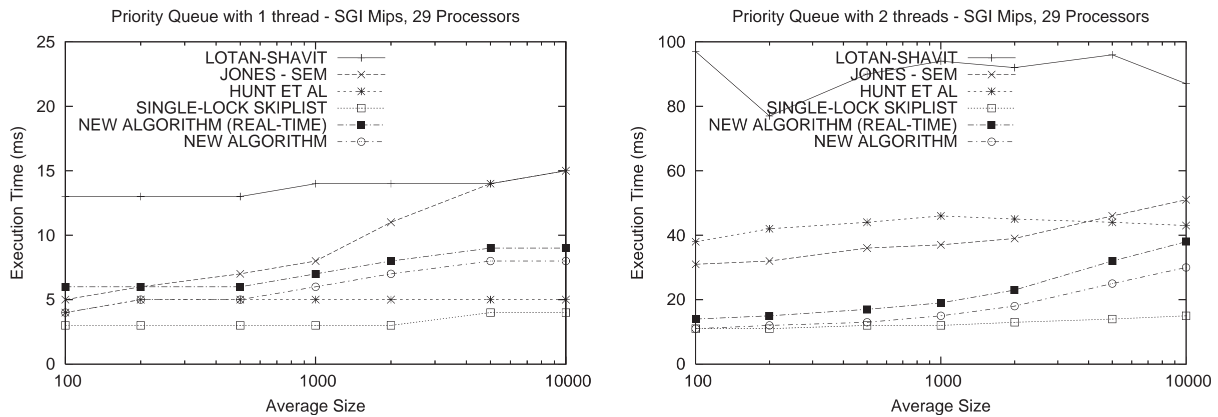

Fig. 15. Experiment with priority queues and no or low concurrency, running with average 100–10,000 nodes.

execution time is drawn as a function of the average number of nodes.

We have also performed experiments with altered distribution of *Insert* operations, varied between 10% upto 90%. For the distribution of 10–40% inserts the dictionary was initialized with 1000 nodes, and for 60–90% inserts the dictionary was initialized as empty. The results from these experiments are shown in Figs. 16 and 17. The average execution time is drawn as a function of the number of threads.

5.3. Results

From the results we can conclude that all of the implementations scale similarly with respect to the average size of the queue. The implementation by Lotan and Shavit [17] scales linearly with respect to increasing number of threads when having full concurrency, although when exposed to preemption its performance is highly dependent on the usage of semaphores, with simple spin-locks the performance decreases very rapidly. The implementation by Hunt et al. [14] shows better but similar behavior for full concurrency. However, it is instead severely punished by using semaphores on systems with pre-emption, because of its built-in contention management mechanism that was designed for simpler locks. The single-lock protected skip list performs better than both Lotan and Shavit and Hunt et al. in all scenarios, using either semaphores or simple spin-locks. The implementation by Jones [16] shows a performance slightly worse or better than the single-lock skip list when exposed to pre-emption, though for full concurrency the performance decreases almost to the implementation by Lotan and Shavit, because of the high overhead connected with semaphores.

Our lock-free implementation scales best compared to all other involved implementations except the single-lock skip list, having best performance already with 3 threads, independently if the system is fully concurrent or involves pre-emptions. Compared to the single-lock skip list, our lock-free implementation performs closely or slightly better for full concurrency, though having rapidly better performance with increasing level of pre-emption. Clearly, our lock-free implementation retains the logarithmic time complexity with respect to the size. However, the normal and real-time versions of our implementation show slightly different behavior, due to several competing time factors. The factors are among many; (i) the real-time version can mark nodes as deleted in parallel, (ii) with larger sizes the accesses get

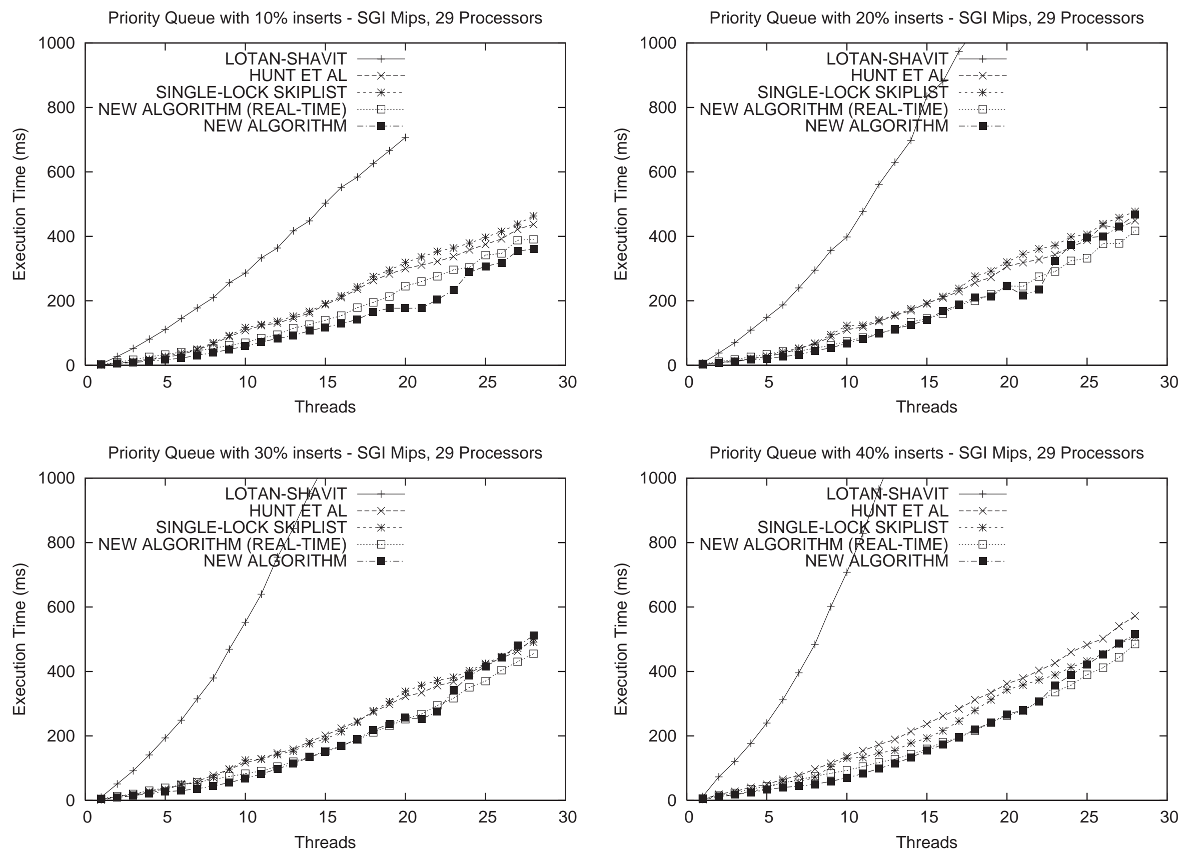

Fig. 16. Experiment with priority queues and full concurrency, varying percentage (between 10 and 40 %) of insert operations, with initial 1000 nodes.

more distributed with resulting lower contention, (iii) the logarithmic time complexity from the nature of the skip list, and (iv) the overhead and contention hot-spot on the shared memory that is connected with the usage of time-stamps in the real-time version.

Even though the implementation by Lotan and Shavit is also based on a skip list with a similar approach as our algorithm, it performs significantly slower, especially on systems with pre-emption. This performance penalty is because of several reasons; (i) there are very many locks involved, each which must be implemented using an atomic primitive having almost the same contention as a CAS operation, (ii) the competition for the locks is comparingly high and increases rapidly with the level in the skip list, with resulting conflicts and waiting on the execution of the blocking operations critical sections (which if possibly pre-empted gets very long), and (iii) the high overhead caused by the garbage collection scheme. The algorithms by Hunt et al. and Jones are also penalized by the drawbacks of many locks, with increasing number of conflicts and blockings with higher level in the tree or heap structure.

For the experiments with altered distribution of *Insert* operations and full concurrency, the hierarchy among the involved implementations are quite different. For 10–40% *Insert* operations the implementation by Hunt et al. shows similar performance as the single-lock skip list and our lock-free implementations, and for 60–90% it shows slightly or significantly better performance. However, neither of the altered scenarios is practically reasonable as long-term scenarios as the priority queues then will have either ever-increasing or almost average zero sizes. An ever-increasing priority queue will be highly impractical in the concern of memory, and for a priority queue with an average size of zero it would suffice with much simpler data structures than skip lists or trees.

6. Extended algorithm

When we have concurrent *Insert* and *DeleteMin* operations we might want to have certain real-time properties of the semantics of the *DeleteMin* operation, as expressed in [17]. The *DeleteMin* operation should only return items that have been inserted by an *Insert* operation that finished before the *DeleteMin* operation started. To ensure this we are adding timestamps to each node. When the node is fully inserted its timestamp is set to the current time. Whenever the *DeleteMin* operation is invoked it first checks the current time, and then discards all nodes that have a timestamp that is after this time. In the code of the implementation (see Figs. 6–8 and 10), the additional statements that involve timestamps are marked within the framed boxes. The

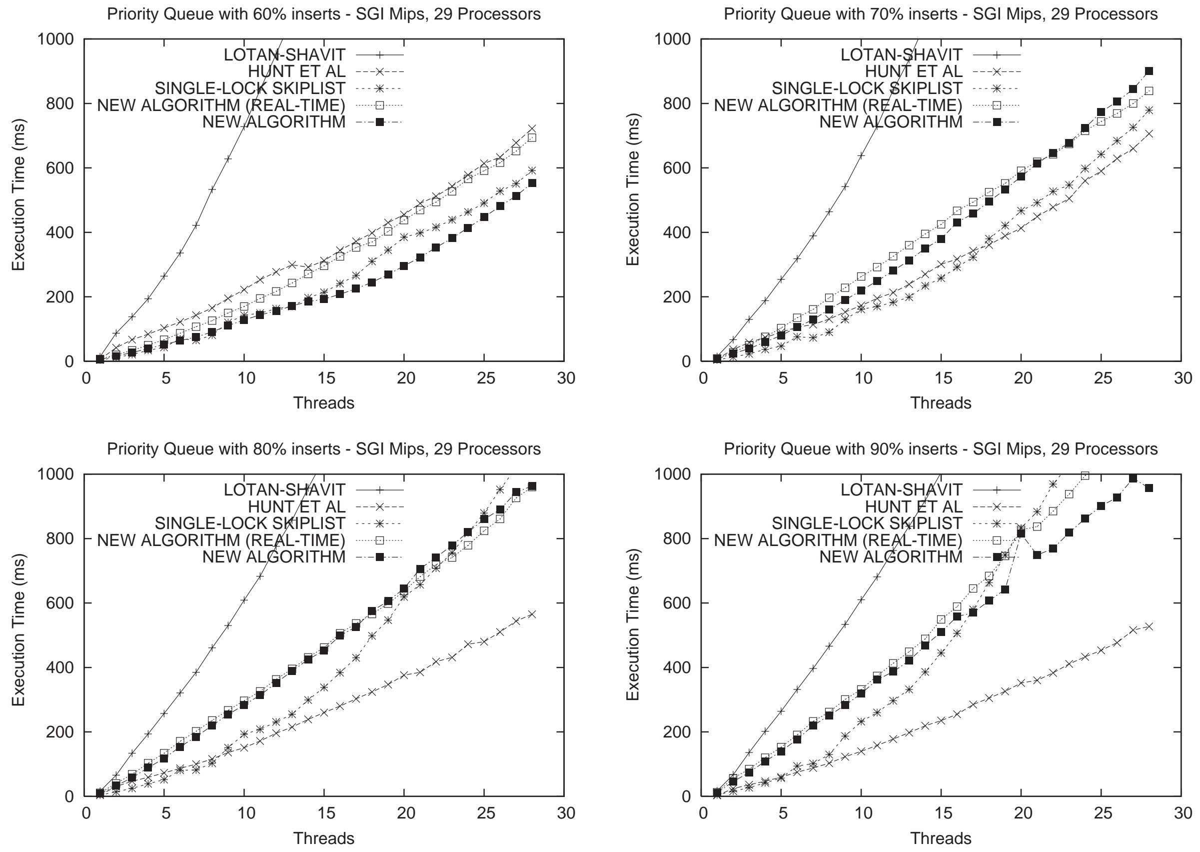

Fig. 17. Experiment with priority queues and full concurrency, varying percentage (between 60 and 90%) of insert operations, with initial 0 nodes.

function *getNextTimeStamp*, see Fig. 21, creates a new timestamp. The function *compareTimeStamp*, see Fig. 21, compares if the first timestamp is less, equal or higher than the second one and returns the values −1,0 or 1, respectively.

As we are only using the timestamps for relative comparisons, we do not need real absolute time, only that the timestamps are monotonically increasing. Therefore we can implement the time functionality with a shared counter; the synchronization of the counter is handled using CAS. However, the shared counter usually has a limited size (i.e. 32 bits) and will eventually overflow. Therefore the values of the timestamps have to be recycled. We will do this by exploiting information that are available in real-time systems, with a similar approach as in [25].

We assume that we have n periodic tasks in the system, indexed $\tau_1 ... \tau_n$. For each task τ_i we will use the standard notations T_i, C_i, R_i and D_i to denote the period (i.e. min period for sporadic tasks), worst case execution time, worst case response time and deadline, respectively. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines, i.e. $\forall i \mid R_i \leqslant D_i$.

For a system scheduled with fixed priority, the response time for a task in the initial system can be calculated using the standard response time analysis techniques [11]. If we with B_i denote the blocking time (the time the task can be delayed by lower priority tasks) and with $hp(i)$ denote the set of tasks with higher priority than task τ_i, the response time R_i for task τ_i can be formulated as

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{9}$$

The summand in the above formula gives the time that task τ_i may be delayed by higher priority tasks. For systems scheduled with dynamic priorities, there are other ways to calculate the response times [11].

Now we examine some properties of the timestamps that can exist in the system. Assume that all tasks call either the *Insert* or *DeleteMin* operation only once per iteration. As each call to *getNextTimeStamp* will introduce a new timestamp in the system, we can assume that every task invocation will introduce one new timestamp. This new timestamp has a value that is the previously highest known value plus one. We assume that the tasks always execute within their response times R with arbitrary many interruptions, and that the execution time C is comparably small. This means that the increment of highest timestamp respective the write to a node with the current timestamp can occur anytime within the interval for the response time. The maximum time for an *Insert* operation to finish is the same as the response time R_i for its task τ_i. The minimum time between two index

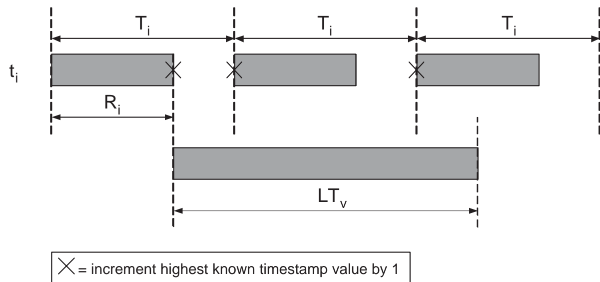

Fig. 18. Maximum timestamp increasement estimation—worst case scenario.

increments is when the first increment is executed at the end of the first interval and the next increment is executed at the very beginning of the second interval, i.e. $T_i - R_i$. The minimum time between the subsequent increments will then be the period T_i. If we denote with LT_v the maximum lifetime that the timestamp with value v exists in the system, the worst case scenario in respect of growth of timestamps is shown in Fig. 18.

The formula for estimating the maximum difference in value between two existing timestamps in any execution becomes as follows:

$$MaxTag = \sum_{i=0}^{n} \left(\left\lceil \frac{\max_{v\in\{0..\infty\}} LT_v}{T_i} \right\rceil + 1 \right). \tag{10}$$

Now we have to bound the value of $\max_{v\in\{0..\infty\}} LT_v$. When comparing timestamps, the absolute value of these are not important, only the relative values. Our method is that we continuously traverse the nodes and replace outdated timestamps with a newer timestamp that has the same comparison result. We traverse and check the nodes at the rate of one step to the right for every invocation of an *Insert* or *DeleteMin* operation. With outdated timestamps we define timestamps that are older (i.e. lower) than any timestamp value that is in use by any running *DeleteMin* operation. We denote with *AncientVal* the maximum difference that we allow between the highest known timestamp value and the timestamp value of a node, before we call this timestamp outdated.

$$AncientVal = \sum_{i=0}^{n} \left\lceil \frac{\max_j R_j}{T_i} \right\rceil. \tag{11}$$

If we denote with t_{ancient} the maximum time it takes for a timestamp value to be outdated counted from its first occurrence in the system, we get the following relation:

$$AncientVal = \sum_{i=0}^{n} \left\lfloor \frac{t_{\text{ancient}}}{T_i} \right\rfloor > \sum_{i=0}^{n} \left(\frac{t_{\text{ancient}}}{T_i} \right) - n, \tag{12}$$

$$t_{\text{ancient}} < \frac{AncientVal + n}{\sum_{i=0}^{n} \frac{1}{T_i}}, \tag{13}$$

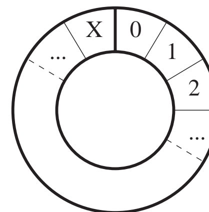

Fig. 19. Timestamp value recycling.

Now we denote with t_{traverse} the maximum time it takes to traverse through the whole list from one position and getting back, assuming the list has the maximum size N.

$$N = \sum_{i=0}^{n} \left\lfloor \frac{t_{\text{traverse}}}{T_i} \right\rfloor > \sum_{i=0}^{n} \left(\frac{t_{\text{traverse}}}{T_i} \right) - n, \tag{14}$$

$$t_{\text{traverse}} < \frac{N + n}{\sum_{i=0}^{n} \frac{1}{T_i}}. \tag{15}$$

The worst case scenario is that directly after the timestamp of one node gets traversed, it gets outdated. Therefore we get

$$\max_{v\in\{0..\infty\}} LT_v = t_{\text{ancient}} + t_{\text{traverse}}. \tag{16}$$

Putting all together we get

$$MaxTag < \sum_{i=0}^{n} \left(\left\lceil \frac{N + 2n + \sum_{k=0}^{n} \left\lceil \frac{\max_j R_j}{T_k} \right\rceil}{T_i \sum_{l=0}^{n} \frac{1}{T_l}} \right\rceil + 1 \right). \tag{17}$$

The above equation gives us a bound on the length of the "window" of active timestamps for any task in any possible execution. In the unbounded construction the tasks, by producing larger timestamps every time they slide this window on the $[0, \ldots, \infty]$ axis, always to the right. The approach now, is instead of sliding this window on the set $[0, \ldots, \infty]$ from left to right, to cyclically slide it on a $[0, \ldots, X]$ set of consecutive natural numbers, see Fig. 19. Now at the same time we have to give a way to the tasks to identify the order of the different timestamps because the order of the physical numbers is not enough since we are re-using timestamps. The idea is to use the bound that we have calculated for the span of different active timestamps. Let us then take a task that has observed v_i as the lowest timestamp at some invocation τ. When this task runs again as τ', it can conclude that the active timestamps are going to be between v_i and $(v_i + MaxTag) \bmod X$. On the other hand, we should make sure that in this interval $[v_i, \ldots, (v_i + MaxTag) \bmod X]$ there are no old timestamps. By looking closer at Eq. (10) we

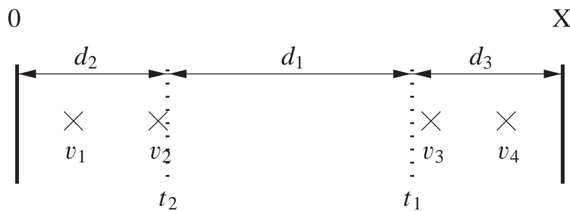

Fig. 20. Deciding the relative order between reused timestamps.

can conclude that all the other tasks have written values to their registers with timestamps that are at most *MaxTag* less than v_i at the time that τ wrote the value v_i. Consequently if we use an interval that has double the size of *MaxTag*, τ' can conclude that old timestamps are all on the interval $[(v_i - MaxTag) \bmod X, \ldots, v_i]$.

Therefore we can use a timestamp field with double the size of the maximum possible value of the timestamp.

$$TagFieldSize = MaxTag * 2,$$

$$TagFieldBits = \lceil \log_2 TagFieldSize \rceil .$$

In this way τ' will be able to identify that v_1, v_2, v_3, v_4 (see Fig. 20) are all new values if $d_2 + d_3 < MaxTag$ and can also conclude that:

$$v_3 < v_4 < v_1 < v_2.$$

The mechanism that will generate new timestamps in a cyclical order and also compare timestamps is presented in Fig. 21 together with the code for traversing the nodes. Note that the extra properties of the priority queue that are achieved by using timestamps are not complete with respect to the *Insert* operations that finishes with an update. These update operations will behave the same as for the standard version of the implementation.

Besides from real-time systems, the presented technique can also be useful in non real-time systems as well. For example, consider a system of $n = 10$ threads, where the minimum time between two invocations would be $T = 10$ ns, and the maximum response time $R = 1{,}000{,}000{,}000$ ns (i.e. after 1 s we would expect the thread to have crashed). Assuming a maximum size of the list $N = 10{,}000$, we will have a maximum timestamp difference $MaxTag < 1{,}000{,}010{,}030$, thus needing 31 bits. Given that most systems have 32-bit integers and that many modern systems handle 64 bits as well, it implies that this technique is practical for also non real-time systems.

```
// Global variables
timeCurrent: integer
checked: pointer to Node
// Local variables
time,newtime,safeTime: integer
current,node,next: pointer to Node

function compareTimeStamp(time1:integer,
 time2:integer):integer
C1    if time1=time2 then return 0;
C2    if time2=MAX_TIME then return -1;
C3    if time1> time2 and (time1-time2)≤ MAX_TAG or
       time1< time2 and (time1-time2+MAX_TIME)
       ≤ MAX_TAG then return 1;
C4    else return -1;

function getNextTimeStamp():integer
G1    repeat
G2       time:=timeCurrent;
G3       if (time+1)≠ MAX_TIME then newtime:=time+1;
G4       else newtime:=0;
G5    until CAS(&timeCurrent,time,newtime);
G6    return newtime;

procedure TraverseTimeStamps()
T1    safeTime:=timeCurrent;
T2    if safeTime ≥ ANCIENT_VAL then
T3       safeTime:=safeTime-ANCIENT_VAL;
T4    else safeTime:=safeTime+MAX_TIME-ANCIENT_VAL;
T5    while true do
T6       node:=READ_NODE(checked);
T7       current:=node;
T8       next:=ReadNext(&node,0);
T9       RELEASE_NODE(node);
T10      if compareTimeStamp(safeTime,
          next.timeInsert)> 0 then
T11         next.timeInsert:=safeTime;
T12      if CAS(&checked,current,next) then
T13         RELEASE_NODE(current);
T14         break;
T15      RELEASE_NODE(next);
```

Fig. 21. Creation, comparison, traversing and updating of bounded timestamps.

7. Conclusions

We have presented a lock-free algorithmic implementation of a concurrent priority queue. The implementation is based on the sequential skip list data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use skip lists for building concurrent priority queues our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous lock-free/wait-free concurrent priority queue algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes skip lists practical: simplicity. Previous lock-free/wait-free algorithms did not perform well because of their complexity; furthermore they were often based on atomic primitives that are not available in today's systems.

We compared our algorithm with some of the most efficient implementations of priority queues known. Experiments show that our implementation scales well, and with 3 threads or more our implementation outperforms the corresponding lock-based implementations, for all cases on both fully concurrent systems as well as with pre-emption.

We believe that our implementation is of highly practical interest for multi-threaded applications. We are currently incorporating it into the NOBLE [26] library.

Acknowledgments

We wish to thank the anonymous reviewers for their helpful comments on improving the presentation of this paper.

This work is partially funded by: (i) the National Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research, and (ii) the Swedish Research Council for Engineering Sciences.

References

[1] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: double-ended queues as an example, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003.

[2] G. Barnes, Wait-free algorithms for heaps, Technical Report, Computer Science and Engineering, University of Washington, February 1992.

[3] D. Detlefs, P. Martin, M. Moir, G. Steele Jr, Lock-free reference counting, in: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, 2001.

[4] M. Fomitchev, Lock-free linked lists and skip lists, Master's thesis, York University, November 2003.

[5] M. Fomitchev, E. Ruppert, Lock-free linked lists and skip lists, in: Proceedings of the 23rd Annual Symposium on Principles of Distributed Computing, 2004, pp. 50–59.

[6] K.A. Fraser, Practical lock-freedom, Ph.D. Thesis, University of Cambridge, Technical Report, No. 579, February 2004.

[7] M. Grammatikakis, S. Liesche, Priority queues and sorting for parallel simulation, IEEE Trans. Software Eng. SE-26 (5) (2000) 401–422.

[8] M. Greenwald, Non-blocking synchronization and system design, Ph.D. Thesis, Stanford University, Palo Alto, CA, 1999.

[9] T.L. Harris, A pragmatic implementation of non-blocking linked lists, in: Proceedings of the 15th International Symposium of Distributed Computing, 2001, pp. 300–314.

[10] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Systems 11 (1) (1991) 124–149.

[11] N. Audsley, A. Burns, R. Davis, K. Tindell, A. Wellings, Fixed priority pre-emptive scheduling: an historical perspective, Real-Time Systems 8 (2/3) (1995) 129–154.

[12] M. Herlihy, V. Luchangco, M. Moir, The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structure, in: Proceedings of 16th International Symposium on Distributed Computing, 2002.

[13] M. Herlihy, J. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Systems 12 (3) (1990) 463–492.

[14] G. Hunt, M. Michael, S. Parthasarathy, M. Scott, An efficient algorithm for concurrent priority queue heaps, Inform. Process. Lett. 60 (3) (1996) 151–157.

[15] A. Israeli, L. Rappoport, Efficient wait-free implementation of a concurrent priority queue, in: Proceedings of the Seventh International Workshop on Distributed Algorithms, Lecture Notes in Computer Science, vol. 725, Springer, Berlin, 1993, pp. 1–17.

[16] D.W. Jones, Concurrent operations on priority queues, Commun. ACM 32 (1) (1989) 132–137.

[17] I. Lotan, N. Shavit, Skiplist-based concurrent priority queues, in: Proceedings of the International Parallel and Distributed Processing Symposium 2000, IEEE Press, New York, 2000, pp. 263–268.

[18] M.M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, 2002, pp. 21–30.

[19] M.M. Michael, M.L. Scott, Correction of a memory management method for lock-free data structures, Technical Report, Computer Science Department, University of Rochester, 1995.

[20] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Commun. ACM 33 (6) (1990) 668–676.

[21] R. Rajkumar, Real-time synchronization protocols for shared memory multiprocessors, in: Proceedings of the 10th International Conference on Distributed Computing Systems, 1990, pp. 116–123.

[22] A. Silberschatz, P. Galvin, Operating system concepts, Addison-Wesley, 1994.

[23] L. Sha, R. Rajkumar, J. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization, IEEE Trans. Comput. 39 (9) (1990) 1175–1185.

[24] N. Shavit, A. Zemach, Scalable concurrent priority queue algorithms, in: Symposium on Principles of Distributed Computing, 1999, pp. 113–122.

[25] H. Sundell, P. Tsigas, Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information, in: Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA 2000), IEEE Press, New York, 2000, pp. 433–440.

[26] H. Sundell, P. Tsigas, NOBLE: A non-blocking inter-process communication library, in: Proceedings of the Sixth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, Lecture Notes in Computer Science, Springer, Berlin, 2002.

[27] H. Sundell, P. Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, Technical Report 2003-01, Computing Science, Chalmers University of Technology, January 2003.

[28] H. Sundell, P. Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, in: Proceedings of the 17th International Parallel and Distributed Processing Symposium, IEEE Press, New York, 2003, p. p. 11.

[29] H. Sundell, P. Tsigas, Scalable and lock-free concurrent dictionaries, in: Proceedings of the 19th ACM Symposium on Applied Computing, ACM Press, New York, 2004, pp. 1438–1445.

[30] P. Tsigas, Y. Zhang, Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors, in: Proceedings of the International Conference on Measurement and Modeling of Computer Systems, ACM Press, New York, 2001, pp. 320–321.

[31] P. Tsigas, Y. Zhang, Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies, in: Proceedings of the 3rd ACM Workshop on Software and Performance, ACM Press, New York, 2002, pp. 55–67.

[32] J.D. Valois, Lock-free data structures, Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.

Håkan Sundell was born in 1968 in Sweden. Since 1983 he has been very active in the home computer and assembly language programming communities with special interest in data compression algorithms and emulation techniques. He received a M.Sc. degree in computer science in 1996 at Göteborg University. Between the years 1995–1999 he worked as a senior consultant and systems programmer within the Swedish tele-communication industry and multimedia community. In 2004 he received a Ph.D. degree in computer science at Chalmers University of Technology. His main research interests are in efficient synchronization for distributed systems and shared data structures.

Philippas Tsigas' research interests include concurrent data structures for multiprocessor systems, communication and coordination in parallel systems, fault-tolerant computing, mobile computing. He received a BSc in Mathematics from the University of Patras, Greece and a Ph.D. in Computer Engineering and Informatics from the same University. Philippas was at the National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present he is an associate professor at the Department of Computing Science at Chalmers University of Technology, Sweden (www.cs.chalmers.se/~tsigas).